CSCE 625 Programming Assignment 1 Report

Peihong Guo

September 19, 2014

1 Introduction

In this project, a equation solver using informed search is implemented. The implementation uses A-star search with a well chosen heuristic, and employs an optimization that significantly speeds up the search process. The solver depends on a set of rules that transform equations or expressions to their less complex equivalence. The rule system is implemented in an extensible way that new rules can be added to the solver to enable more mathematical manipulations. The solver is able to simplify a wide range of equations with various degree of complexity, which will be demonstrated in the results.

2 Algorithm

The solver consists of two main modules, the search module and the rule system module. The search algorithm is a variation of the standard A-star algorithm, where optimizations are included to speed up the search process. The search is performed on the set of equivalent equations $S = \{e_i\}$ defined by a set of transformation rules R. The is a graph search problem because the each equation in S can be transformed to and from a few other equations in S.

2.1 Search

A-star algorithm is guaranteed to reach an optimal goal state given a admissible and consistent heuristic. Therefore one of the most important aspect of the search module is the definition of an effective heuristic.

Heuristic The heuristic is inspired from the solution process of an equation: the ultimate goal of solving an equation is to move *every* component of the equation to the right hand side *except for* the unknown. The number of actions needed to achieve this goal is at least the depth of the unknown in the prefix tree of the equation. For example, consider the following equation:

$$x + 1 = 2$$

Its prefix representation is

$$(= \ (+ \ x \ 1) \ 2)$$

The depth of the unknown x is 1 (in the tree of LHS of the equation), and we need exactly 1 action to solve it. Another example:

$$x = y - 1/2 * x$$

Its prefix representation is

$$(= \ x \ (- \ y \ (* \ (/ \ 1 \ 2) \ x)$$

The depth of the unknown x is 0 in the LHS tree and 2 in the RHS tree, and we need at least 3 actions to solve it. Note that in this case, the unknow also appears in the RHS, and we need at least 1 extra action to move it to LHS. This is why the number of actions is at least 3 rather than 2.

To state the above observation more formally, the heuristic is defined as:

$$h = VariableDepth(LHS) + VariableDepth(RHS) + VariableIn(RHS)?1 : 0$$

The extra 1 added to the heuristic is critical to search performance. In fact, this slight modification leads to at least 2X speed up. The rational behind this is equations with the unknown in RHS would definitely require at least one more action to move the unknow to the left hand side, even if the equation is already solved (say, $1 = x$).

From the analysis above, it is apparent this heuristic is admissible. On the other hand, the heuristic is also consistent because the transformation from one equation to another could increase the variable depth for at most one(details about this are discussed in the rules section), thus the heuristic satisfies:

$$h(x) <= d(x, y) + h(y)$$

if distance between two states is defined as the number of actions between them.

One problem of this heuristic is that it does not consider the complexity of the expressions without the unknown. Consider the following example:

$$x = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$$

The heuristic value by the previous definition is 0 here, but it takes at least 15 steps to achieve the solution state. For such cases, the previously defined heuristic significantly underestimates the distance of a equation to the final solution.

From the equation tree it is obvious that the above example has a depth of 15 on the RHS:

$$(= \ (+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1(+ \ 1 \ (+ \ 1(+ \ 1 \ 1)))))))))))))))$$

This leads to a similar heuristic that works very well for such cases: the maximum depth of a expression. This heuristic is defined as:

$$h = MaxDepth(LHS) + MaxDepth(RHS) + VariableIn(RHS)?1 : 0$$

However, this heuristic is not admissible.

With the heuristic defined above, the search algorithm still takes a long time to simplify some very simple equations. A brief analysis reveals that the cause of this is problem. For non-trivial equations, mostly equations with more than 10 nodes, the possible states to examine explodes quickly as the search progresses. This is expected because there are exponential number of possible states, and the branching factor is very high (at least ten in most cases). The overwhelming number of states to examine leads to a unbearable long running time. To alleviate this, a restarting scheme is included in the search algorithm.

Restarting The restarting scheme, as its name indicates, is a method of restarting the search at some point of the search process in order to achieve faster running time. The basic idea is that most states generated in the early stage of the search is useless because they either have could large distance to goal state and still capable of generating large number of states with little value. Meanwhile, some states are worth more focus because they are significantly simplified compared to the original equation. However, such states could be placed in the middle of the search queue due to a high path cost. Restart the search process with such states is thus a good idea to remove all the useless early-stage states as well as to focus the precious computation power on the most promising candidates. That is, the search process is restarted *every time* a *good enough* candidate is found.

It is vital to define *good enough* candidates properly because the restarting is radical pruning scheme that discards all previously discovered search nodes, which could be dangerous if the candidate chosen as restart point is no better than the previous starting point. The following criteria is used when determining if a candidate is significantly better than the previously found best solution:

Denote the new candidate as e_{new} and the previous best solution as e_{best},

- If LHS of e_{new} is the unknown and the unknown is not in its RHS, and e_{best} has the unknown in both side or the LHS of e_{best} is not exactly the unknown, e_{new} is better; and vice versa. For example:

  $$x = 4/2$$

  is better than

  $$x = 4 - x \text{ or } 2 * x = 4$$

- If the number of leaf nodes in the equation tree of e_{new} is smaller than that in e, e_{new} is better; and vice versa. For example:

  $$x = 2$$

  is better than

  $$x = 1 + 1$$

- If the heuristic value of e_{new} is smaller than that of e, e_{new} is better; and vice versa. For example:

  $$x = 1$$

is better than

$$1 = x$$

- If the number of variable of e_{new} is smaller than that of e, e_{new} is better; and vice versa. For example:

  $$2 * x = 2$$

  is better than:

  $$x + x = 2$$

- If the number of leaf nodes in the LHS of e_{new} is smaller than that of e, e_{new} is better; and vice versa. For example:

  $$x = 1/2$$

  is better than

  $$2 * x = 1$$

- If by all the above criteria, e_{new} and e are equally good, e is better.

The restarting scheme is very powerful. It reduces the running time of the search by a factor of 10 when handling complex equations.

Node Expansion Node expansion is the process of transforming an equation to an equivalent one. In the search algorithm, node expansion is done by applying all available rules to *every* applicable node in the equation tree. For example, applying commutativity to $(x + 1) + 2 = 4$ results in two new equations: $(1 + x) + 2 = 4$ and $2 + (x + 1) = 4$. This approach usually lead to a very high branching factor for complicated equations where a single rule can apply to several places at the same time, not to mention there could be multiple rules applicable to the equation.

2.2 Rules

The rule system is the other important module in the solver. It is built in a generic way that managing rules is extremely simple. There are two types of rules in the rule system: evaluation rules and transformation rules.

The evaluation rules are implemented in a per-case basis because they all involve actual mathematical computation. For example, $1 + 1$ will be evaluated directly and the node (+ 1 1) will be replaced with a single node 2 in the expression/equation tree.

Transformation rules are implemented to be generic so that the application of these rules can be handled in the same way. To define a transformation rule, one just write down the equation before and after applying the rule as well as all the symbols in the rule. The defined rules are then parsed into expression/equation trees and used for pattern matching and node modification during the search process. Because of the unified representation, the application of these rules are simplified and can be handled uniformly.

For a list of all rules included in the system, please refer to Appendix. **??**.

Representation A rule is defined as a tuple $R(s, p, m)$, where s is the set of symbolic elements in the rule, p is the pattern of the rule and m is the modifier of the rule(the expression/equation after applying the rule). For example, the associativity of addition is defined as follows (in Python code):

```
ass_add = Pattern(['a', 'b', 'c'], '(a+b)+c', 'a+(b+c)')
ass_add_mns = Pattern(['a', 'b', 'c'], '(a+b)-c', '(a-c)+b')
```

Application To apply a rule to transform an equation, a breath first traversal is used to visit every node in the equation tree. When visiting a node, the rule being considered is used to match the node. If the matching succeeds, the node will be replaced with a node generated with the modifier of this rule, and the entire equation is then transformed. The modified equation is treated as a variation of the current equation and will be enqueued to the search frontier.

When matching a node n with a pattern, the pattern expression tree p is used where each node n_p in the tree p is compared to a corresponding node in the subtree induced by n. The comparisons are considered successful if the nodes are the same or n_p is a symbolic node(a node whose content is a symbol defined by the rule). For example, $x + 1$ will be matched by the commutativity rule `R(['a', 'b'], 'a+b', 'b+a')` where x is matched with a and 1 is matched with b. This produces the modified node $1 + x$ because the modifier is `'b+a'`.

3 Results

The following equations are used for testing the solver:

(1)$x = 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1$

(2)$x = (2 + 10) * (2^2)$

(3)$x = 6 * 2/(-1 + 4 * 0 + 1)$

(4)$y|x = y - x$

(5)$(2 * \text{sqrt}(x) * 3) - y = pi$

(6)$x * 3 * y * 4 * z * 5/6 = 800$

(7)$e^x = z * (\sin(y)^2 + \cos(y)^2)$

(8)$e^x = \sin(8 + 3/2 * z + y - 1/2 * z)^2 + \cos(y + 8 + z)^2$

(9)$\text{Diff}(x^2 + 10 * x + 2, x) = 4 * z$

For equation (4), it means the variable to solve for is y rather than x, and the — is a separator.

The running times for solving the equations above are listed in Table.1. It is clear that H_1, the maximum depth heuristic, works very well for equation 1. However, its performance is not as good when applied to more complex equations

(Equation 5, 6 and 8) where the depth of variable is a more appropriate heuristic. For those equations, H_2 performs much better.

Also from the results we can see that H_2 without adding one when the unknown appears in RHS runs from 1.3X to 5X slower than H_2, while H_2 without restart runs at least 10X slower and even fail for sufficiently complicated equations such as Equation 1, 6 and 8.

Equation	$0.5 * (H_1 + H_2)$	H_1	H_2	H_2 without +1	H_2 without Restart
1	0.1810	0.1807	3.5069	4.0323	-
2	0.0030	0.0027	0.0046	0.0051	0.0042
3	0.0069	0.0067	0.0169	0.0062	0.0235
4	0.0122	0.0103	0.0098	0.0099	0.0120
5	0.1034	0.1469	0.0656	0.0727	0.7449
6	4.3405	1.3226	0.2259	1.1154	-
7	0.0147	0.0141	0.0225	0.0203	-
8	-	-	66.2701	89.9225	-
9	0.5881	1.1775	0.5852	0.9391	-

Table 1: Running time (in seconds) of different heuristics / optimizations. H_1 is the max depth heuristic, and H_2 is the max variable depth heuristic. H_2 without +1 is the H_2 without adding one when the unknown appears in RHS. A missing value means the search failed for that case.

A Implemented Rules

A.1 Evaluation Rules

1. Addition: $a + b$
2. Subtraction: $a - b$
3. Multiplication: $a * b$
4. Division: a/b
5. Power: a^b
6. Root: $root(a, b) = a^{1/b}$
7. Negation: $-a$
8. Function evaluations: $\sin, \cos, \tan, \text{asin}, \text{acos}, \text{atan}, \text{sqrt}, \log, \ln$
9. Differentiation: $\text{Diff}(a, x) = 0$

A.2 Transformation Rules

Move Rules

1. $a = b \rightarrow b = a$
2. $a + b = c \rightarrow a = c - b$
3. $a - b = c \rightarrow a = c + b$
4. $a * b = c \rightarrow a = c/b$
5. $a/b = c \rightarrow a = c * b$
6. $a^b = c \rightarrow a = root(c, b)$

Inverse Function Rules

1. $\text{sqrt}(x) = y \rightarrow x = y^2$
2. $\sin(x) = y \rightarrow x = \text{asin}(y)$
3. $\cos(x) = y \rightarrow x = \text{acos}(y)$
4. $\tan(x) = y \rightarrow x = \text{atan}(y)$
5. $\text{asin}(x) = y \rightarrow x = \sin(y)$
6. $\text{acos}(x) = y \rightarrow x = \cos(y)$
7. $\text{atan}(x) = y \rightarrow x = \tan(y)$
8. $\log(x) = y \rightarrow x = 10^y$
9. $\ln(x) = y \rightarrow x = e^y$
10. $e^x = y \rightarrow x = \ln(y)$
11. $10^x = y \rightarrow x = \log(y)$

Associativity Rules

1. $(a + b) + c \rightarrow a + (b + c)$
2. $(a + b) - c \rightarrow (a - c) + b$
3. $(a - b) - c \rightarrow (a - c) - b$
4. $a - (b - c) \rightarrow (a - b) + c$
5. $a - (b + c) \rightarrow (a - b) - c$
6. $(a * b) * c \rightarrow a * (b * c)$

7. $(a * b)/c \rightarrow a * (b/c)$
8. $a * (b/c) \rightarrow (a * b)/c)$
9. $a/(b/c) \rightarrow a/(b * c)$
10. $a/(b/c) \rightarrow (a/b) * c)$
11. $a/(b * c) \rightarrow (a/b)/c)$
12. $(a^b)^c \rightarrow a^{b*c}$

Commutativity Rules

1. $a + b \rightarrow b + a$
2. $a * b \rightarrow b * a$

Distribution Rules

1. $a * (b + c) \rightarrow a * b + a * c$
2. $a * (b - c) \rightarrow a * b - a * c$
3. $(a + b)/c \rightarrow a/c + b/c$
4. $(a - b)/c \rightarrow a/c - b/c$
5. $a * b + a * c \rightarrow a * (b + c)$
6. $a * b - a * c \rightarrow a * (b - c)$
7. $a/b + a/c \rightarrow (a + b)/c$
8. $a/b - a/c \rightarrow (a - b)/c$
9. $a + a * b \rightarrow a * (1 + b)$
10. $a + a \rightarrow a * 2$
11. $a - a * b \rightarrow a * (1 - b)$
12. $a * b - a \rightarrow a * (b - 1)$
13. $a - a \rightarrow 0$

0 1 Rules

1. $a + 0 \rightarrow a$
2. $a - 0 \rightarrow a$
3. $a * 0 \rightarrow 0$
4. $a^0 \rightarrow 1$
5. $a * 1 \rightarrow a$
6. $a/1 \rightarrow a$
7. $a^1 \rightarrow a$

Identities

1. $\sin(x)^2 + cos(x)^2 \rightarrow 1$
2. $e^{a*x} * e^{b*x} \rightarrow e^{(a*x+b*x)}$

Differentiation

1. $\text{Diff}(x, x) \rightarrow 1$
2. $\text{Diff}(x^y, x) \rightarrow y * x^(y - 1)$
3. $\text{Diff}(x + y, z) \rightarrow Diff(x, z) + Diff(y, z)$
4. $\text{Diff}(x - y, z) \rightarrow Diff(x, z) - Diff(y, z)$
5. $\text{Diff}(x * y, z) \rightarrow Diff(x, z) * y + x * Diff(y, z)$
6. $\text{Diff}(x/y, z) \rightarrow (Diff(x, z) * y - x * Diff(y, z))/(y * y)$

Special Rules

1. $1/(1/x) \rightarrow x$
2. $x^a * x^b \rightarrow x^{a+b}$